# Sistemi 2

/ radice.

/bin e /sbin comandi essenziali per l’amministrazione

/boot file per il boot di sistema

/dev file speciali che corrispondono a dispositivi…

Parliamo dell’idea di disciplina di linea: abbiamo detto che ogni applicazione è collegata a un terminale. Questo è abbastanza vero, ma non al 100%, per esempio se così non fosse non sarebbe possibile effettuare la cancellazione di caratteri.  
Tra il processo e il terminale scemo, c’è infatti la disciplina di linea. Se scrivo cat e poi provo a inserire uno standard input, l’input arriva solo dopo che si preme invio. Questa è la disciplina di linea di default, la si può cambiare con “stty -icanon” (così la si toglie), con “stty icanon” la si reimposta. Togliere la disciplina di linea può servire per implementare su terminale qualsiasi programma che non vuole dover aspettare che l’utente prema invio per ricevere un input (magari si vuole giocare a tetris su terminale).  
Col tempo sul terminale è diventato possibile utilizzare i colori, per cambiare colore bisogna usare escape seguito da un comando particolare, per esempio ‘\x1b[38;5;123mciao’ ha \x1b che è l’escape e [38;5;123m è la codifica del colore. C’è da dire inoltre che perché i caratteri speciali siano interpretati bisogna usare l’opzione (-e) dopo il comando (quindi ad esempio echo -e ‘\x1b[38;5;123mciao’ stampa un ciao in azzurro).

Altre sequenze di escape permettono di spostare il cursore.

Quando si dà un’occhiata a tutti i file (con ls -l), in genere la shell colora di blu le cartelle, e altre cose di altri colori. Se ls non pensa di star mandando il messaggio a un terminale non mette i colori, a meno che non glielo si specifichi con l’opzione --color

Come utilizzare la shell in maniera interattiva? In questa modalità essa stampa un prompt (=una sequenza di caratteri che notifica l’essere pronta a ricevere istruzioni). Il prompt è configurabile cambiando la variabile PS1 (con PSI=’nuovo\_prompt’)

Quando la shell riceve una stringa di comandi, come prima cosa usa gli spazi per spezzarla, a questo punto può considerare/tradurre gli alias (utilizzabili per compattare formule lunghe), dopodiché fa delle espansioni (basandosi su espressioni regolari: per esempio “ls \*a.c” vuol dire fai l’ls di tutti i file che iniziano con qualsiasi carattere e terminano con a.c) -> le espansioni vengono fatte dalla shell e non dentro al programma chiamato (ciò è molto più risparmioso), almeno su Linux (windows è terribile come al solito).  
Dopodiché vengono effettuate i dirottamenti input/output (se ci sono il carattere > o < ), dunque il comando scrive verso il suo standard output, ma se c’è un dirottamento il comando scriverà sul file scelto senza rendersene conto (perché per lui quello sarà momentaneamente lo standard output).

Fatto ciò, viene eseguito il “comando”, che è tipicamente un eseguibile/uno script esterno (ma può essere una funzione o un comando build-in). Per vedere se un comando è interno o esterno basta usare il comando “type” seguito dal comando.

La shell va a cercare i comandi dalla variabile d’ambiente $PATH: in questa variabile sono contenuti tutti i percorsi dei file che contengono i comandi chiamabili sulla shell.

Un file può venire trovato nella directory corrente, ma se e solo se tale directory è presente nel PATH, altrimenti è necessario chiamarlo con ./ (questo è fatto per evitare di lanciare per sbaglio comandi “maligni” che hanno lo stesso nome di comandi build-in e che sono nascosti in cartelle scaricate).

Mettendo & dopo un comando molto lungo si può continuare a usare il terminale mentre il processo resta attivo in background: ciò è utile se si redirige l’output verso qualche altra parte.

I comandi più importanti sono:

man (utile per i comandi esterni)

help (da usare per i comandi build-in)

Un sito utile per comprendere che fanno i comandi della shell è “explainshell.com”

Per colorare si può usare “Bat” che è la versione colorata di Cat. C’è anche il Generic Colourizer per aggiungere colori agli output dei comandi.

I permessi in linux si interpretano così: r = read, w = write, prima colonna = proprietario, seconda colonna = membri del gruppo (?), terza colonna = gli altri.

Prima si è detto che la shell spezza sui blank, quindi se si volesse mandare uno spazio bisognerebbe usare il backslash (escape) , oppure utilizzare le doppie virgolette per delimitare la stringa (però a quel punto se si volessero usare le doppie virgolette servirebbe fare l’escape di quelle).  
Si possono usare anche i singoli apici. Dentro la shell ci sono delle variabili e noi possiamo espandere (tradurre) il valore di queste variabili, ma ciò avviene SOLO se per delimitare la stringa usiamo i doppi apici e non i singoli apici (con i singoli apici non avviene l’espansione). La stessa differenza si applica con le espressioni regolari (gli asterischi \* non sono tradotti all’interno dei singoli apici).   
Si può far espandere una variabile delimitandola con le graffe {}.

Per forzare l’espansione si possono usare anche l’opzione -e oppure il prefisso $ prima della stringa delimitata da singoli apici.

Se si vuole cercare nell’history dei comandi, si può usare il comando “history” e si può usare !seguito dal numero del comando, oppure dall’inizio del comando in questione (ti recupera le compilazioni fatte dieci anni prima).

Se si scrive A=10 si sta assegnando alla variabile A il valore 10. Se si scrive A = 10 la shell proverà a eseguire il comando A con per argomenti = e 10.

Fra le variabili comode c’è il $$ = PID della shell stessa. C’è anche il $? Che dà il codice di uscita del comando precedente (se è andato a buon fine sarà 0). Questo può essere utile per quando lanciamo degli script che devono controllare se è fallito qualcosa.

Le variabili della shell non vengono passate ai comandi che lanciamo, a meno che non le definiamo come variabili d’ambiente. Per definire una variabile come d’ambiente basta mettere un export davanti all’assegnazione di valore della variabile (export A=10).

Quando noi modifichiamo le variabili d’ambiente, le modifiche rimangono solo nella shell in cui le abbiamo apportate (a meno che non si apra una shell nuova direttamente da quella in cui abbiamo apportato la modifica).

Se vogliamo che gli export vengano eseguiti ogni volta che apriamo una shell dobbiamo aggiungerli a uno script. Tipicamente ~/.bashrc, ma anche in .[bash]profile.

shopt -s autocd mette l’auto-cd

Se si prova a stampare una variabile che non esiste la shell di base stampa la stringa vuota, ma con set -o nounset viene invece lanciato un errore.

Altri tipi di espansione sono la tilde ~, che indica la home corrente o se seguita da un utente stampa la home di quell’utente.

Se si mettono dei valori tra virgole all’interno delle graffe preceduti da testo e seguiti da altro testo i due testi verranno stampati tante volte quanto il numero di argomenti nelle graffe con in mezzo ogni volta un elemento diverso.

$(( )) ci permette di valutare espressioni matematiche all’interno della shell. Può essere utile per gli script.

$(comando) fa sì che l’output di un comando possa essere utilizzato come parametro per un altro comando. Esempio:  
./args $(ls)

C’è una “sintassi antica” che consiste nel mettere il comando da usare come parametro tra “backticks”.

Il pipe manda l’output di un programma come gli args del main di un altro.

Ci sono due file speciali che possono tornare utili:

/dev/null -> tutto ciò che c’è lì dentro viene buttato via

/dev/zero ->contiene infiniti zeri

Veniamo allo scripting. La shell può eseguire non solo semplici comandi, ma anche sequenze, cicli ecc. A tal proposito la pipeline serve a comporre un comportamento elaborato partendo da comandi semplici (per esempio se voglio contare i file in una cartella posso fare ls -l | wc -l, così facendo wc conta le linee che ls stamperebbe e restituisce quindi il numero dei file). Si può anche riordinare un output e mandarlo in input da qualche altra parte con due pipes e un sort in mezzo.

La pipeline è più furba che salvare i risultati in file e leggerli: innanzitutto con la pipeline i comandi vengono lanciati contemporaneamente, ma soprattutto se il secondo comando ha bisogno dell’output del primo questo parte a utilizzarlo nel momento in cui ne inizia ad arrivare (senza dover aspettare che finisca). Infine, la pipeline non riempie la memoria del sistema, come invece potrebbe fare il creare in continuazione file per salvare outputs.

Per mandare più comandi insieme (ma non in pipeline) si può usare il ; per separarli. Se si separano i comandi con &, questi vengono lanciati in parallelo in background (quindi non si sa quale dei due finirà per primo).

Se io delimito delle serie di comandi (separati da ; ) con le parentesi tonde (), posso fare un dirottamento che abbia effetto su entrambi i comandi contemporaneamente.

A differenza dei linguaggi di programmazione, per la bash 0 è True, mentre qualsiasi valore diverso da 0 è False; si è fatto così perché 0 è il valore che i programmi ritornano quando va tutto bene.

&& e || funzionano come AND e XOR e usano i valori di ritorno dei comandi per decidere se eseguire quelli successivi:  
Esempio:

comando1 && comando2 //comando2 è eseguito solo se comando1 ritorna 0

comando1 || comando2 //comando2 è eseguito solo se comando1 ritorna non 0

Tipicamente i comandi possono essere interrotti premendo ctrl+C. Ciò avviene perché il kernel invia un interrupt di altissimo livello (corrisponde al comando SIGNINT). Per inviare segnali, si può usare il comando kill, tuttavia il processo non muore necessariamente, perché il processo può catturare dei segnali e ripartire (Di base corrisponde al comando SIGTERM(15), che può essere catturato): se lo si chiama col -9 si lancia SIGKILL(9) anziché SIGTERM e questo comando non può essere bloccato.

Si possono creare anche delle funzioni con function name {cmd-list}.

Il comado return può essere usato come in C; local dichiara variabili locali.

Un esempio malvagio può essere la fork bomb : () { : | : &};: